CLOSURES & FUNCTIONS

CS3342

DAVE THOMAS
@PRAGDAVE

BINDINGS (AGAIN)

These two variables have the same name but are different variables

Just this variable

```
a = 123
print(a)

adder = fn (a, b) {
  print(a)
  a + b
}

print(adder(4,5))

print(a)
```

a ⇒ 123

a ⇒ 4

a ⇒ 123

```
fib = fn (n) {
  print(n)
  if n < 2 { n }
  else     { fib(n-1) + fib(n-2) }
}

fib(4)
```

```
Print: 4
Print: 3
Print: 2
Print: 1
Print: 0
Print: 1
Print: 2
Print: 1
Print: 0
```

Even though the call **fib(n-1)** changes **n** in the called function, it doesn't change in the caller

HOW IS THIS DONE?

YOU CHOOSE

(BASED ON SEMANTICS YOU WANT)

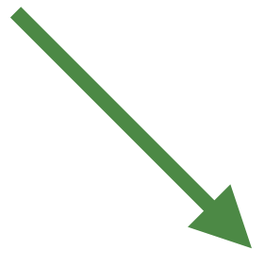
Photo by KT on Unsplash

```
a = 9

adder = fn (b) { a + b }

adder(1)
```

can a function access variables in the scope where it is defined?

no

bindings are local to function execution



yes

bindings implement **closures**

BINDINGS ARE LOCAL TO FUNCTION EXECUTION

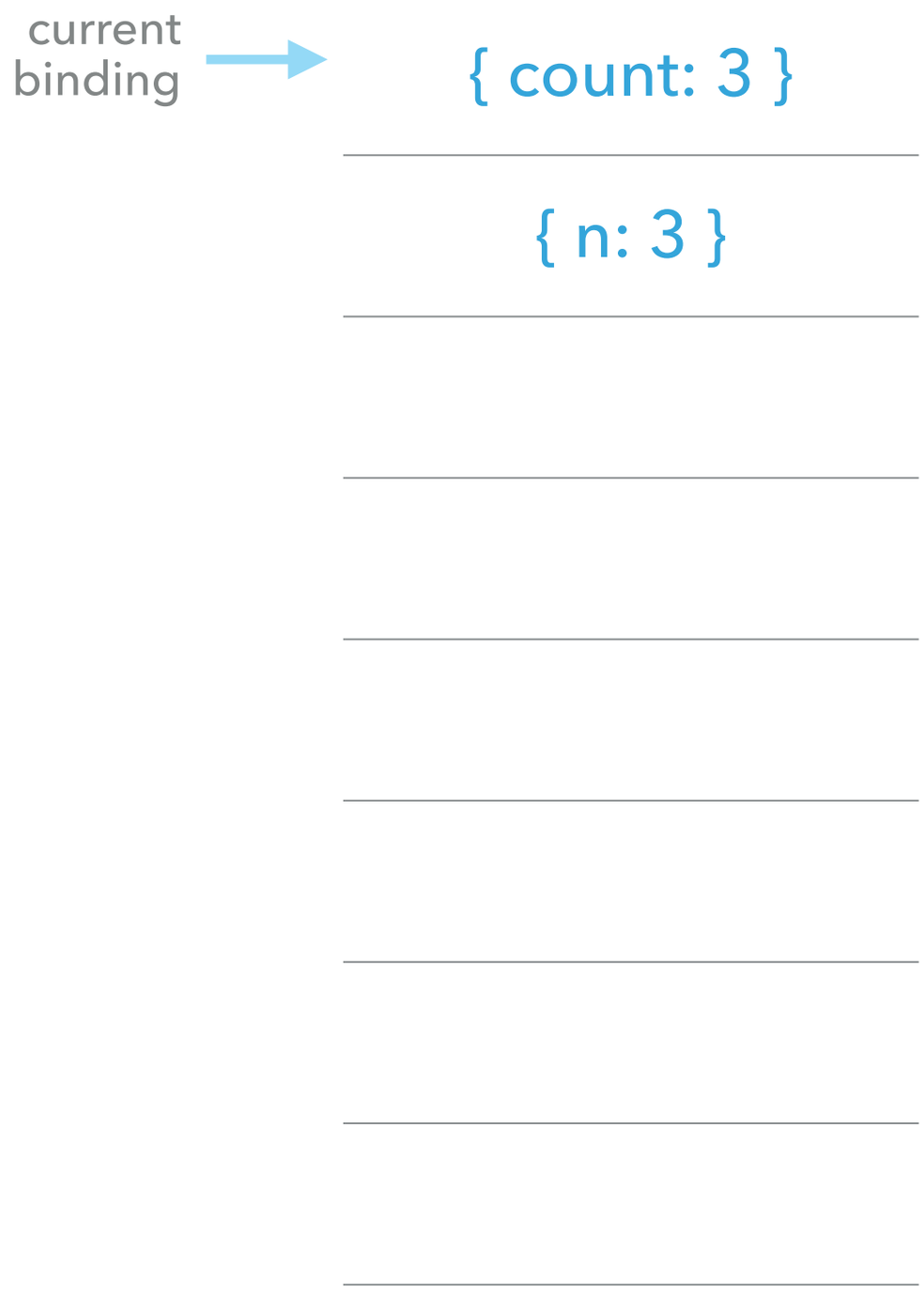
- This is the traditional case (C, Java, etc)
- Maintain a stack of current bindings
- When a function is **called**, a new current binding is created
  - This contains the function's parameters and their values, along with function-local variables
  - This binding (and possibly a global binding) is the only one available for variable lookup.
  - When the function exits, the stack is popped, and the binding is lost

```
count = 3

def fib(n):
  if n < 2:
    return n
  else:
    return fib(n-1) + fib(n-2)

fib(count)
```

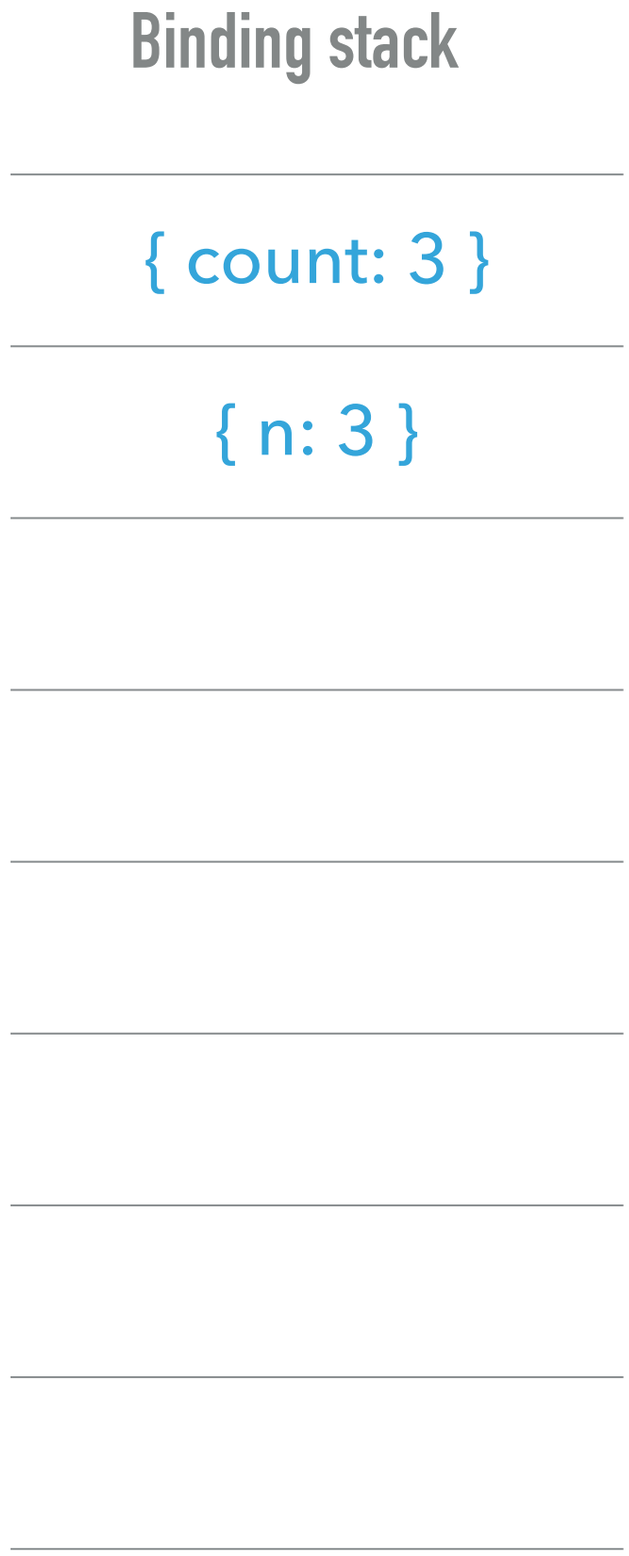
Binding stack

{ count: 3 }

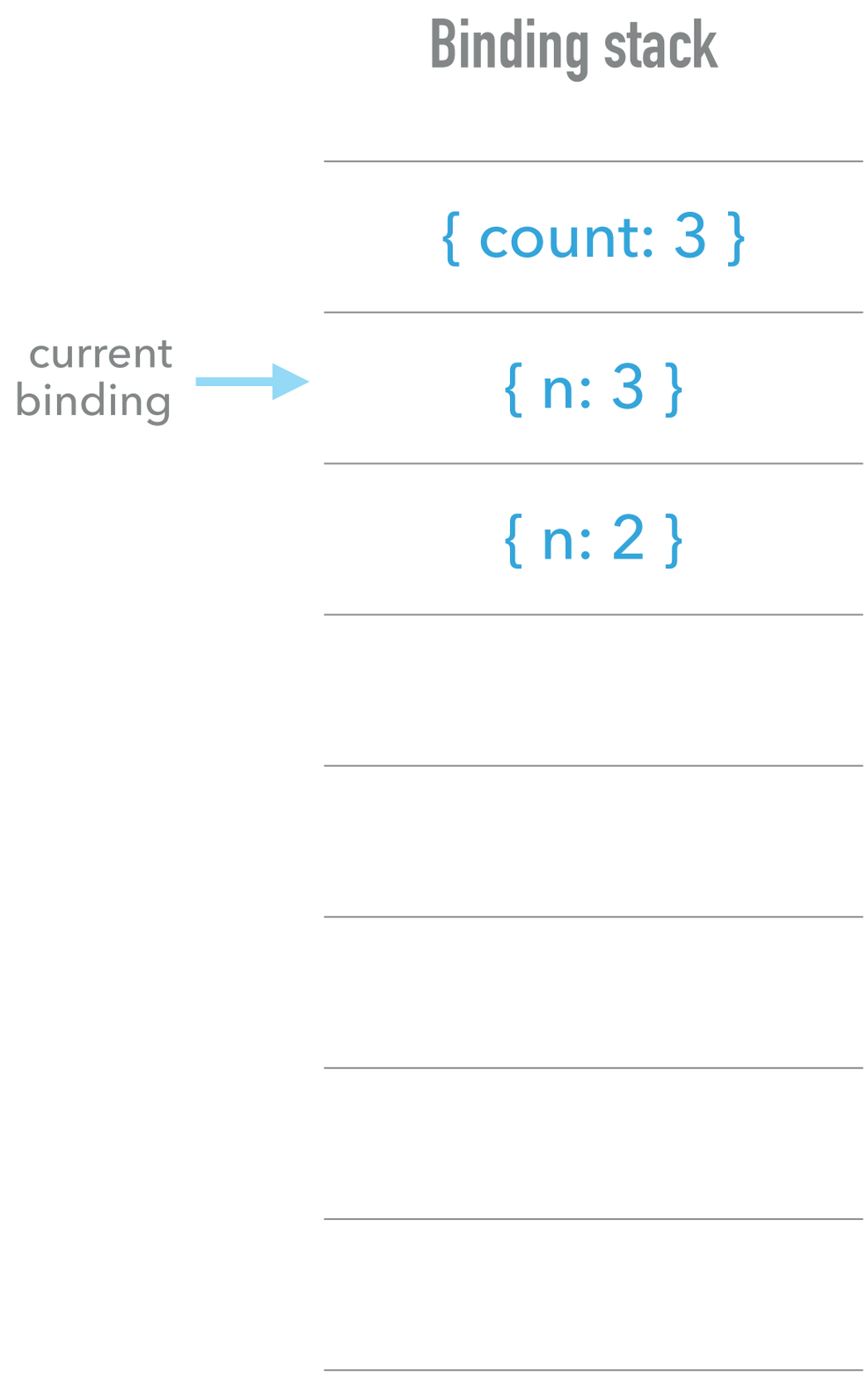
```
count = 3

def fib(n):
  if n < 2:
    return n
  else:
    return fib(n-1) + fib(n-2)

fib(count)
```



```
count = 3

def fib(n):
  if n < 2:
    return n
  else:
    return fib(n-1) + fib(n-2)

fib(count)
```



```
count = 3

def fib(n):
  if n < 2:
    return n
  else:
    return fib(n-1) + fib(n-2)

fib(count)
```



```
count = 3

def fib(n):
  if n < 2:
    return n
  else:
    return fib(n-1) + fib(n-2)

fib(count)
```

current binding

Binding stack

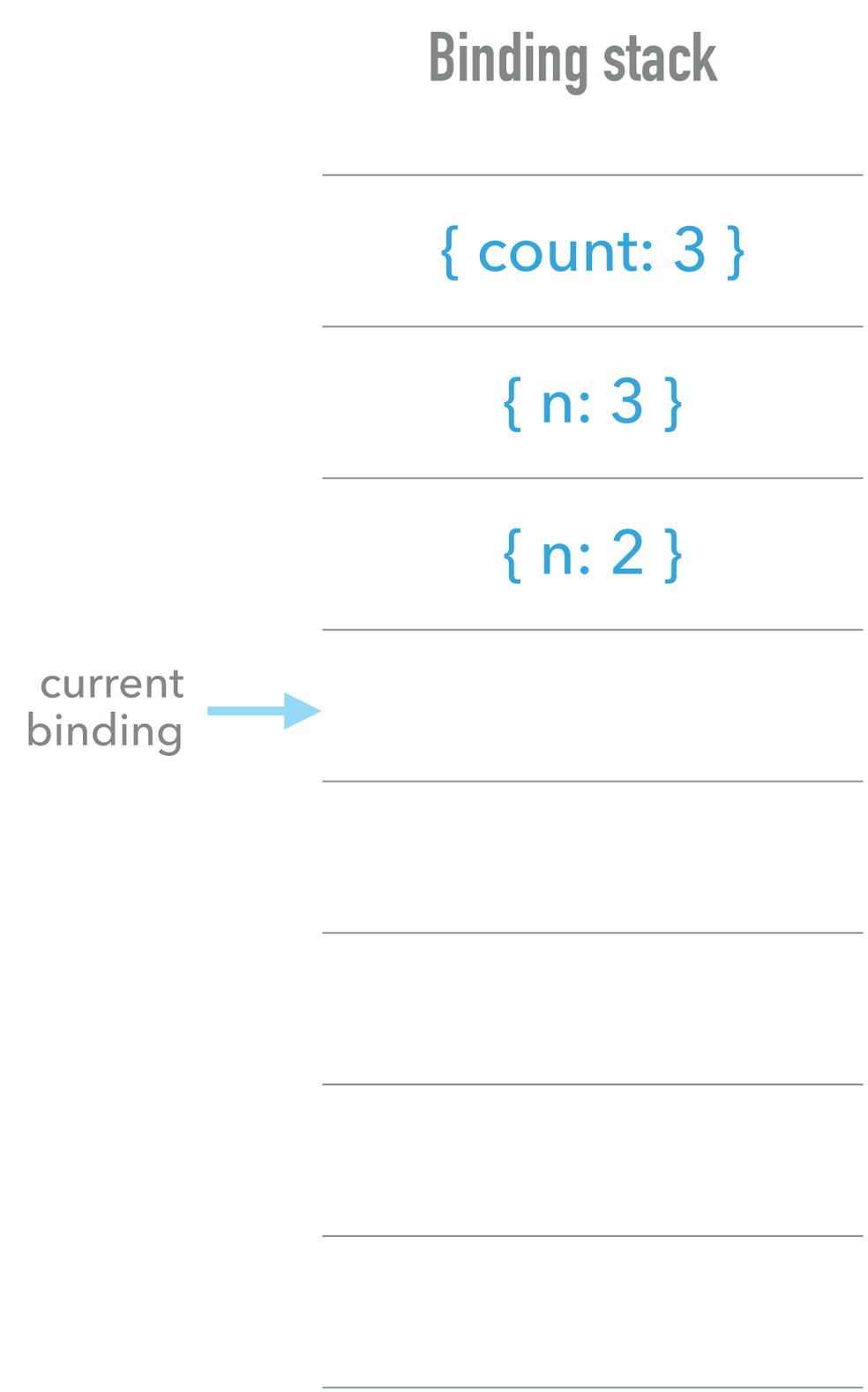
{ count: 3 }

{ n: 3 }

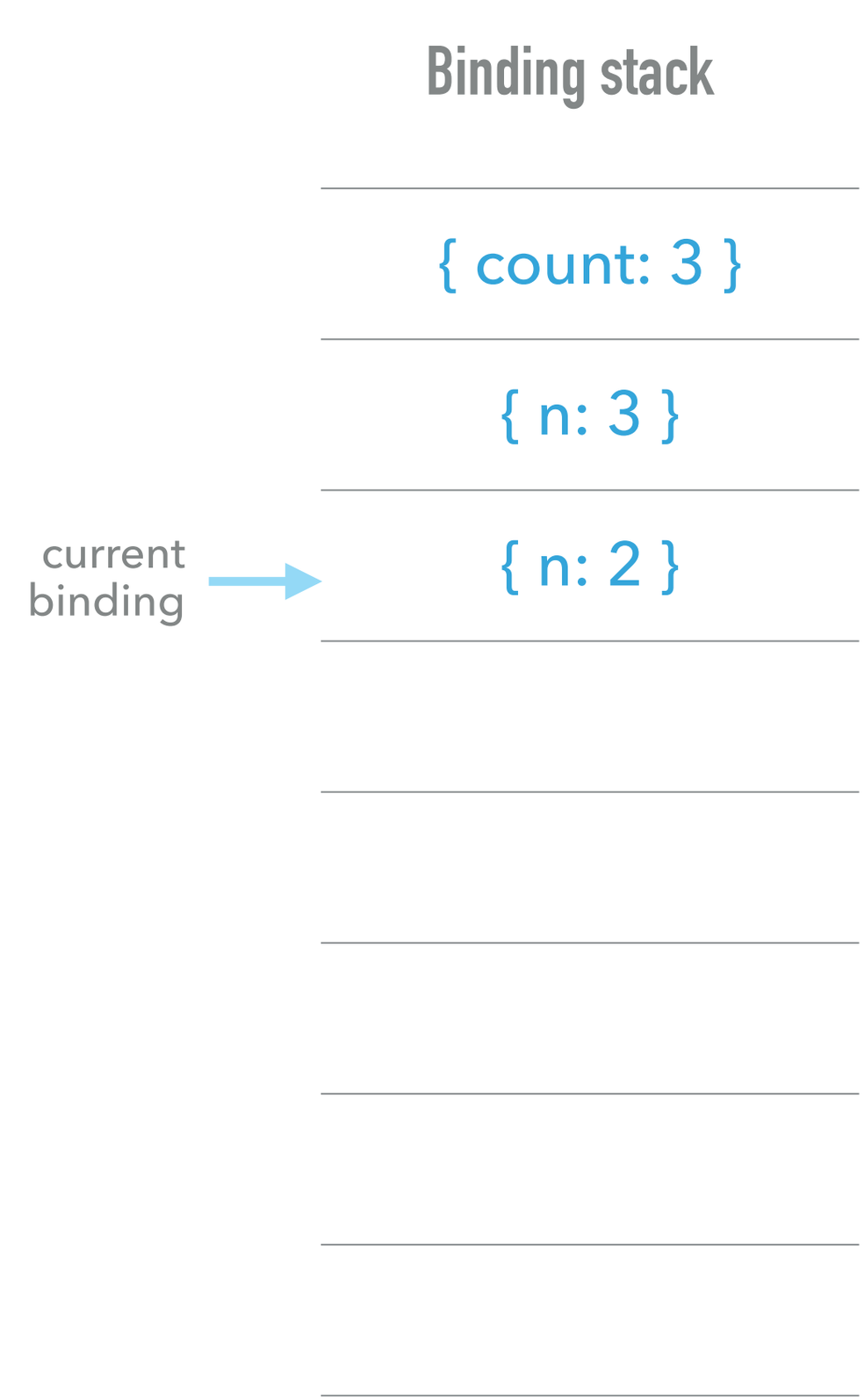
{ n: 2 }

{ n: 1 }

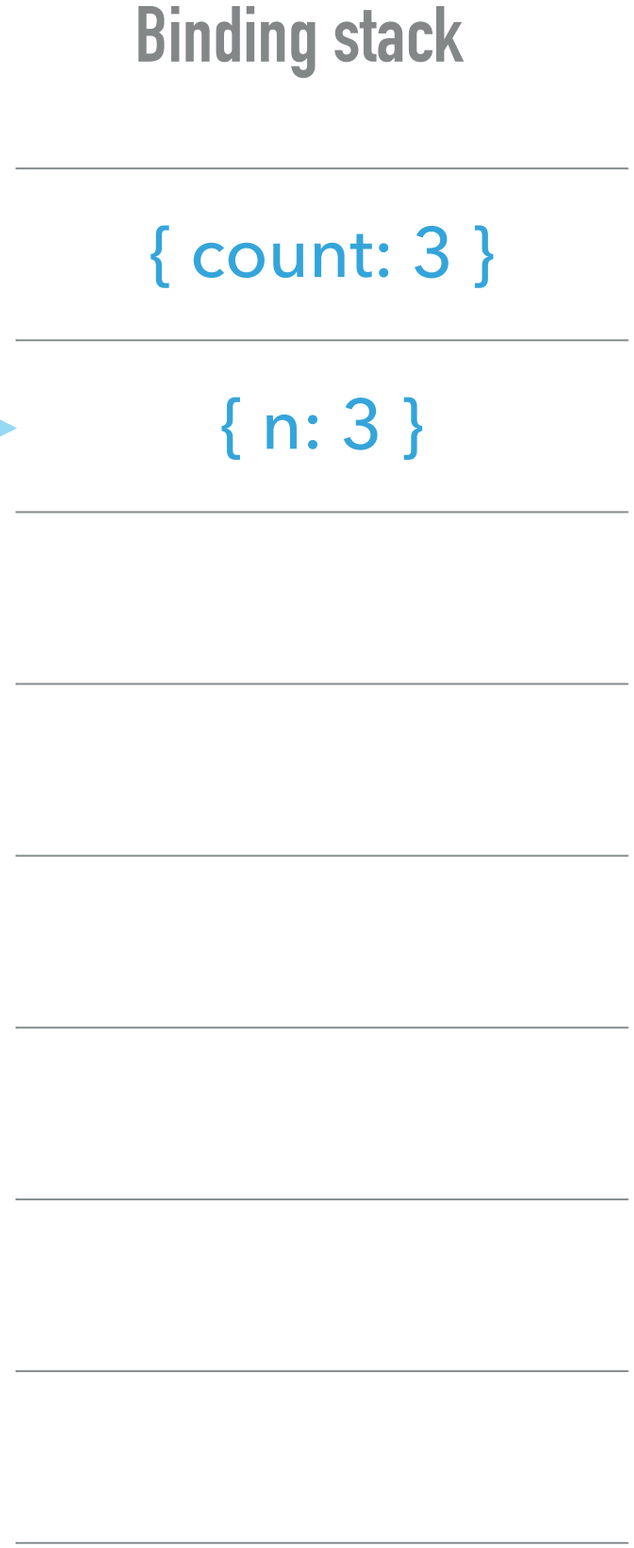
```
count = 3

def fib(n):
  if n < 2:
    return n
  else:
    return fib(n-1) + fib(n-2)

fib(count)
```



```
count = 3

def fib(n):
  if n < 2:
    return n
  else:
    return fib(n-1) + fib(n-2)

fib(count)
```

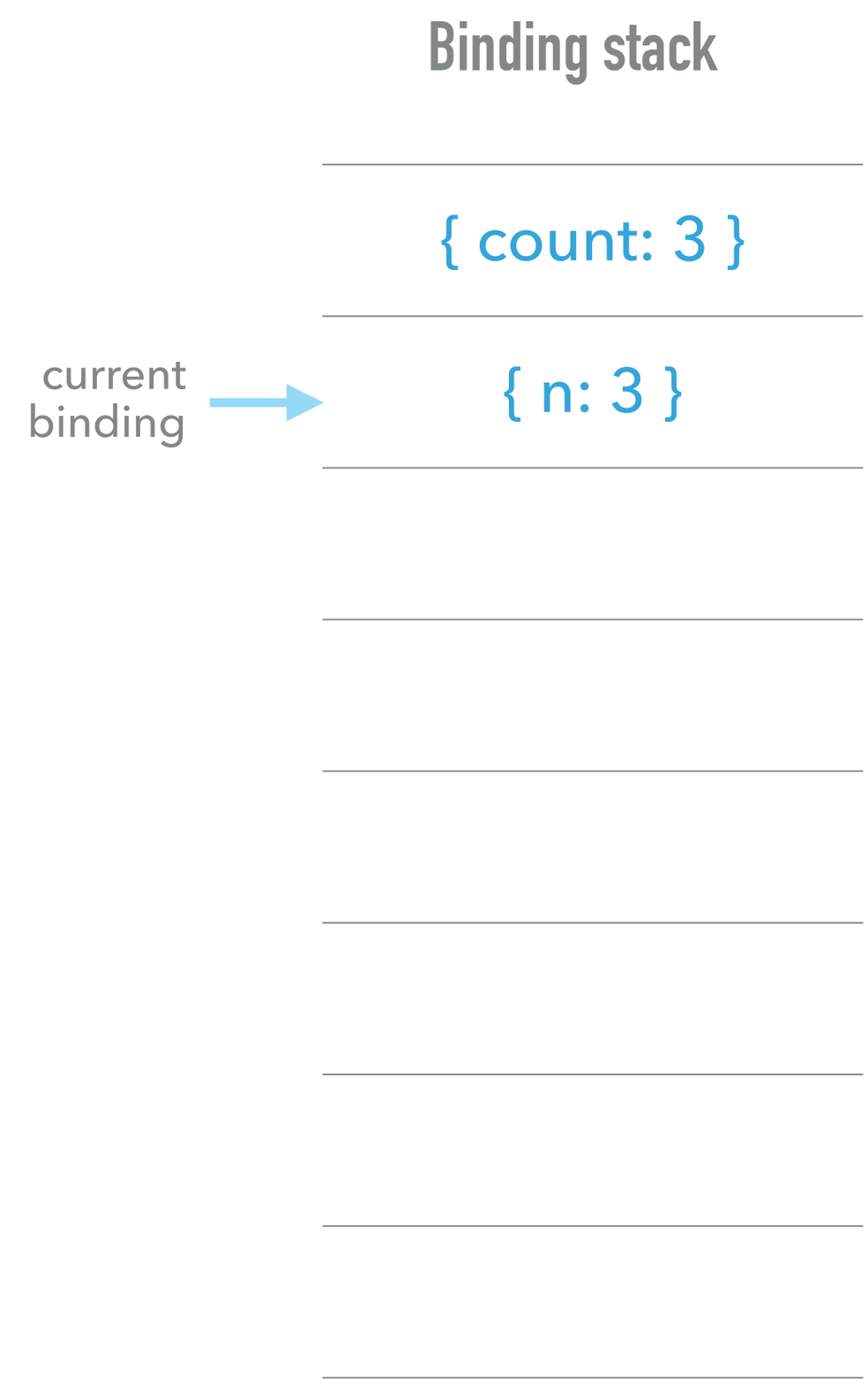


```
count = 3

def fib(n):
  if n < 2:
    return n
  else:
    return fib(n-1) + fib(n-2)

fib(count)
```

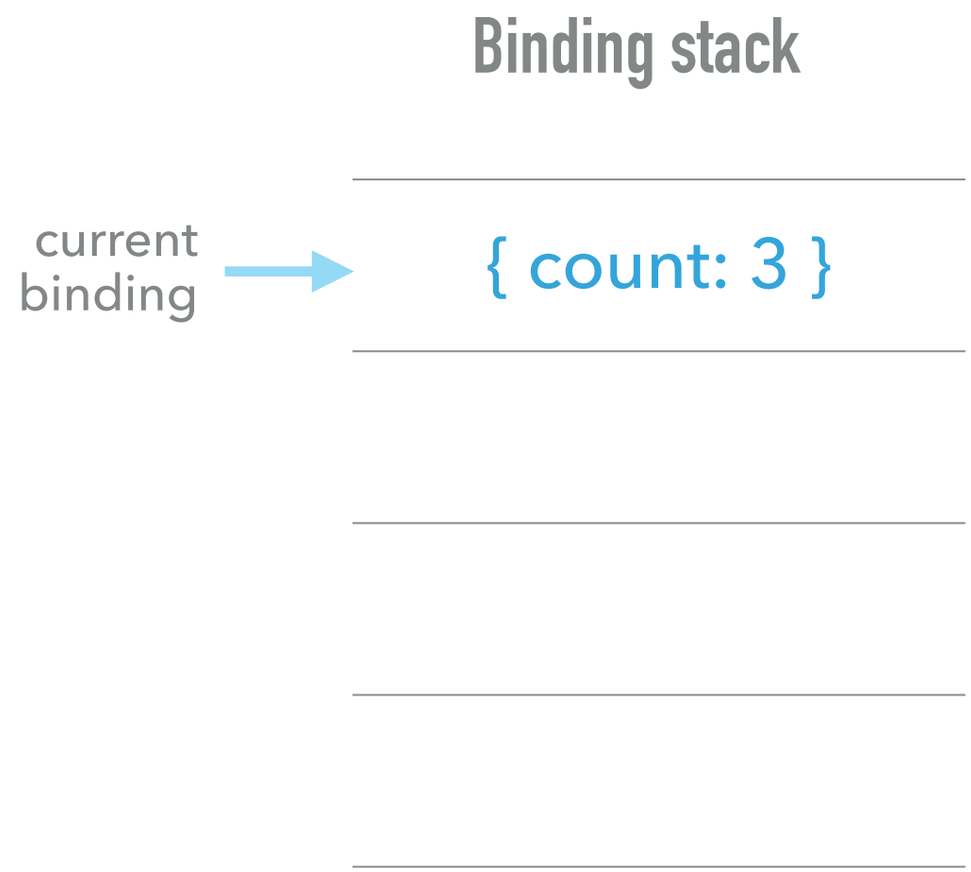


```
count = 3

def fib(n):
  if n < 2:
    return n
  else:
    return fib(n-1) + fib(n-2)

fib(count)
```

```
count = 3

def fib(n):
  if n < 2:
    return n
  else:
    return fib(n-1) + fib(n-2)

fib(count)
```



```
count = 3

def fib(n):
  if n < 2:
    return n
  else:
    return fib(n-1) + fib(n-2)

fib(count)
```



IN PRACTICE

IN PRACTICE

- In interpreters, stack is often handled by having the binding be a local variable in a recursive call
- In non interpreted languages, parameters are pushed onto the stack along with the return address and space for local variables.
- Potential problem if code returns a reference to a variable on the stack

YOU CHOOSE

(BASED ON SEMANTICS YOU WANT)

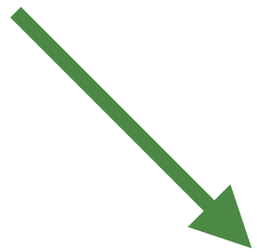
Photo by KT on Unsplash

```
a = 9

adder = fn (b) { a + b }

adder(1)
```

can a function access variables in the scope where it is defined?

no

bindings are local to function execution



yes

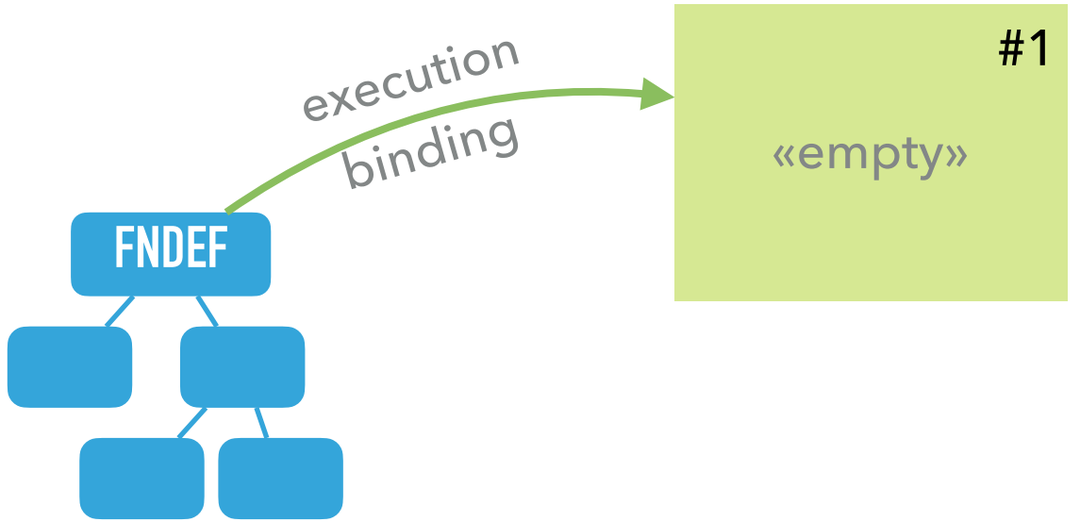
bindings implement **closures**

BINDINGS ARE CLOSURES

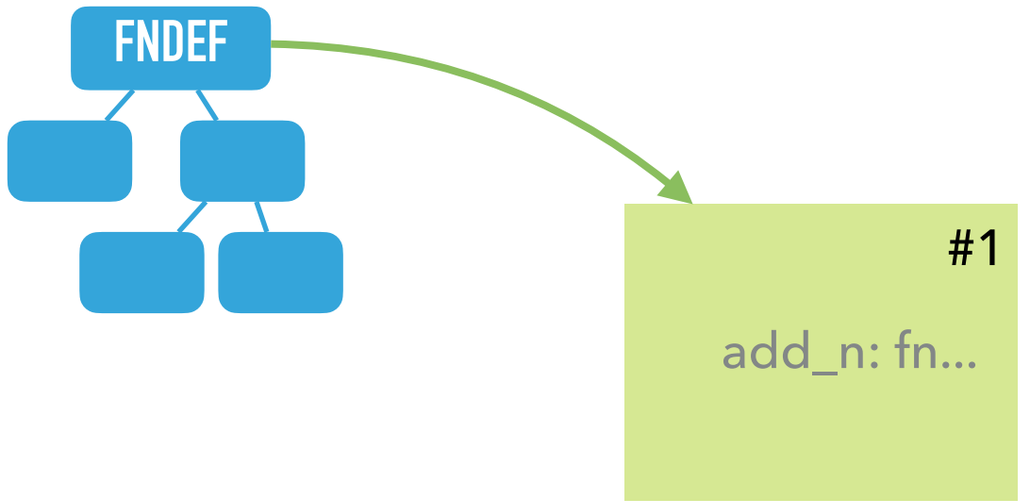
- This is the norm in functional languages, and in languages with anonymous functions
- Bindings now stored in a tree. There's always a current binding.
- When a variable is looked up, look for it in the current binding, then in its parent, then its parent, etc
- When a function is **created**, it carries with it the value of the current binding at point of creation

```
let add_n = fn (n) {
  fn (x) {
    x + n
  }
}


let add_2 = add_n(2)

let add_3 = add_n(3)

print(add_2(5))

print(add_3(5))
```

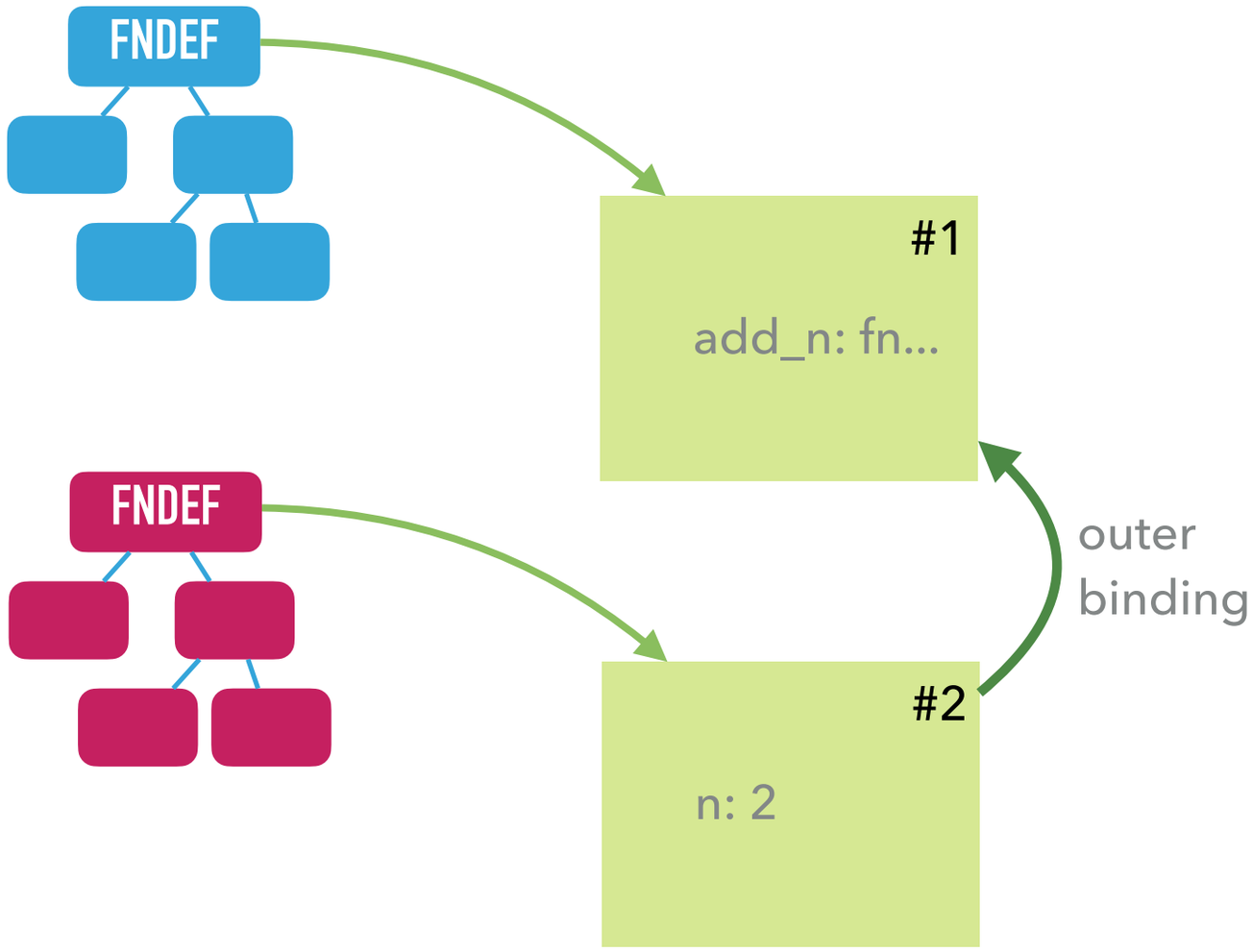
```
let add_n = fn (n) {
  fn (x) {
    x + n
  }
}


let add_2 = add_n(2)

let add_3 = add_n(3)

print(add_2(5))

print(add_3(5))
```



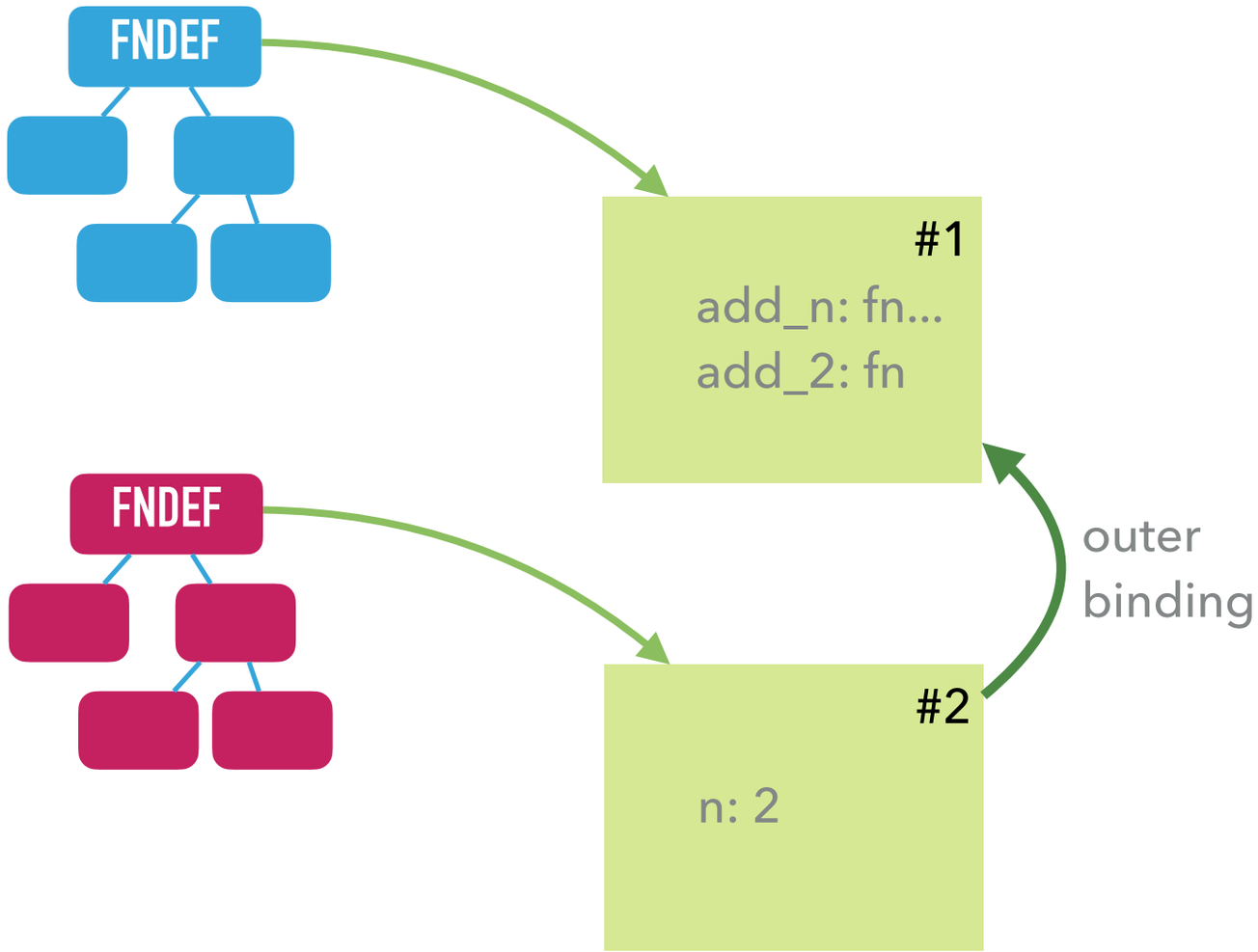
```
let add_n = fn (n) {
  fn (x) {
    x + n
  }
}


let add_2 = add_n(2)

let add_3 = add_n(3)

print(add_2(5))

print(add_3(5))
```



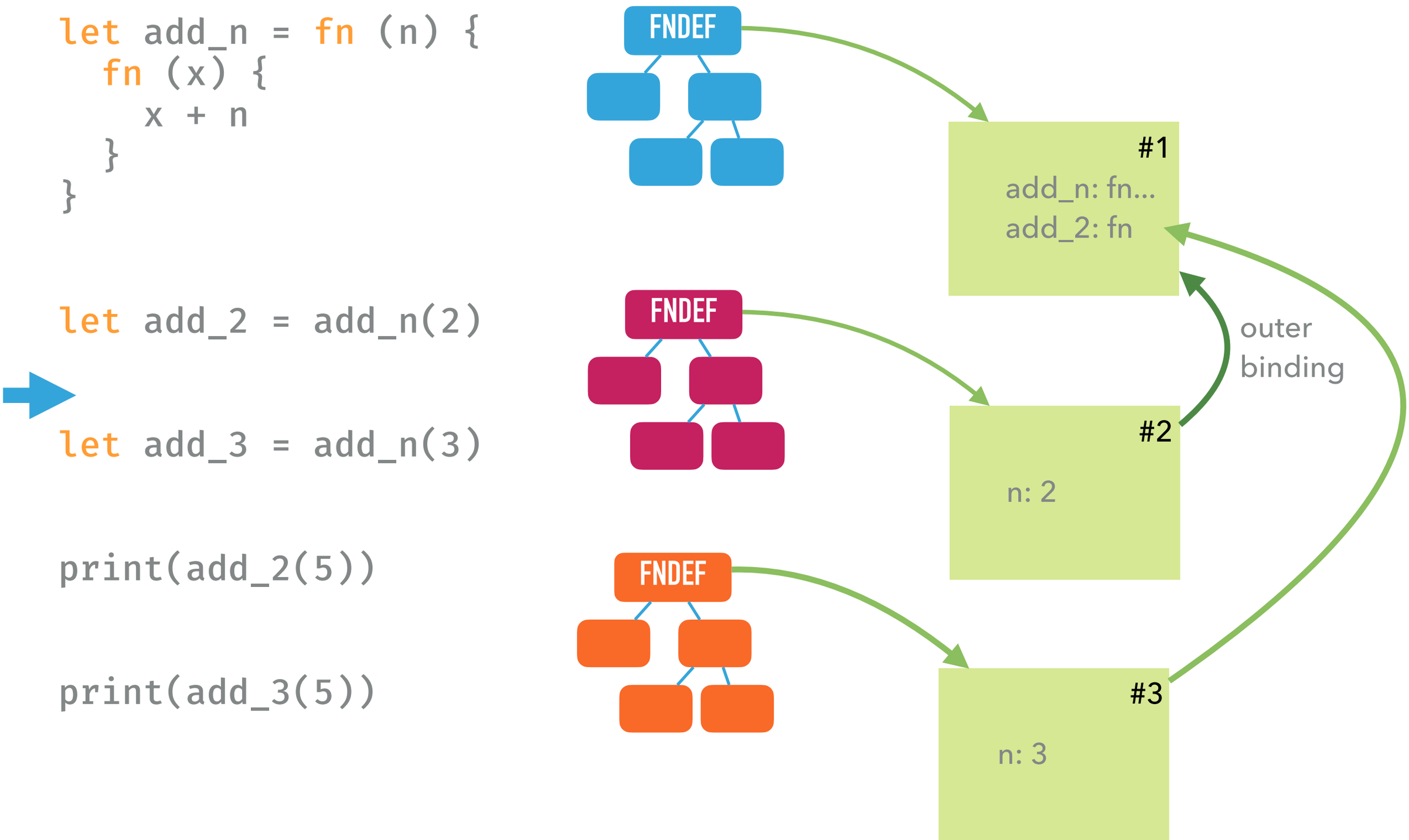
```
let add_n = fn (n) {
  fn (x) {
    x + n
  }
}

let add_2 = add_n(2)

let add_3 = add_n(3)

print(add_2(5))

print(add_3(5))
```



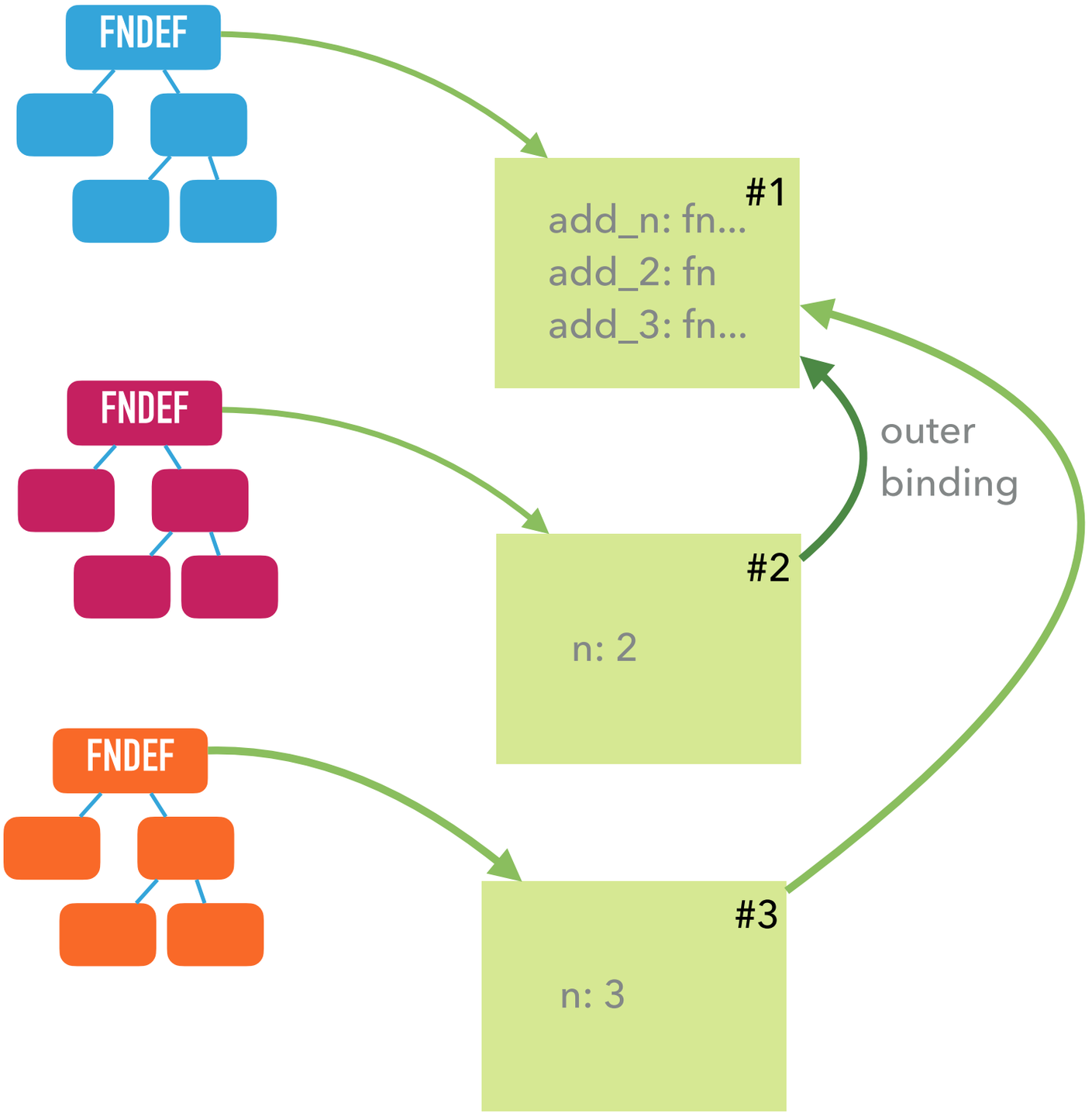
```
let add_n = fn (n) {
  fn (x) {
    x + n
  }
}

let add_2 = add_n(2)

let add_3 = add_n(3)

print(add_2(5))

print(add_3(5))
```

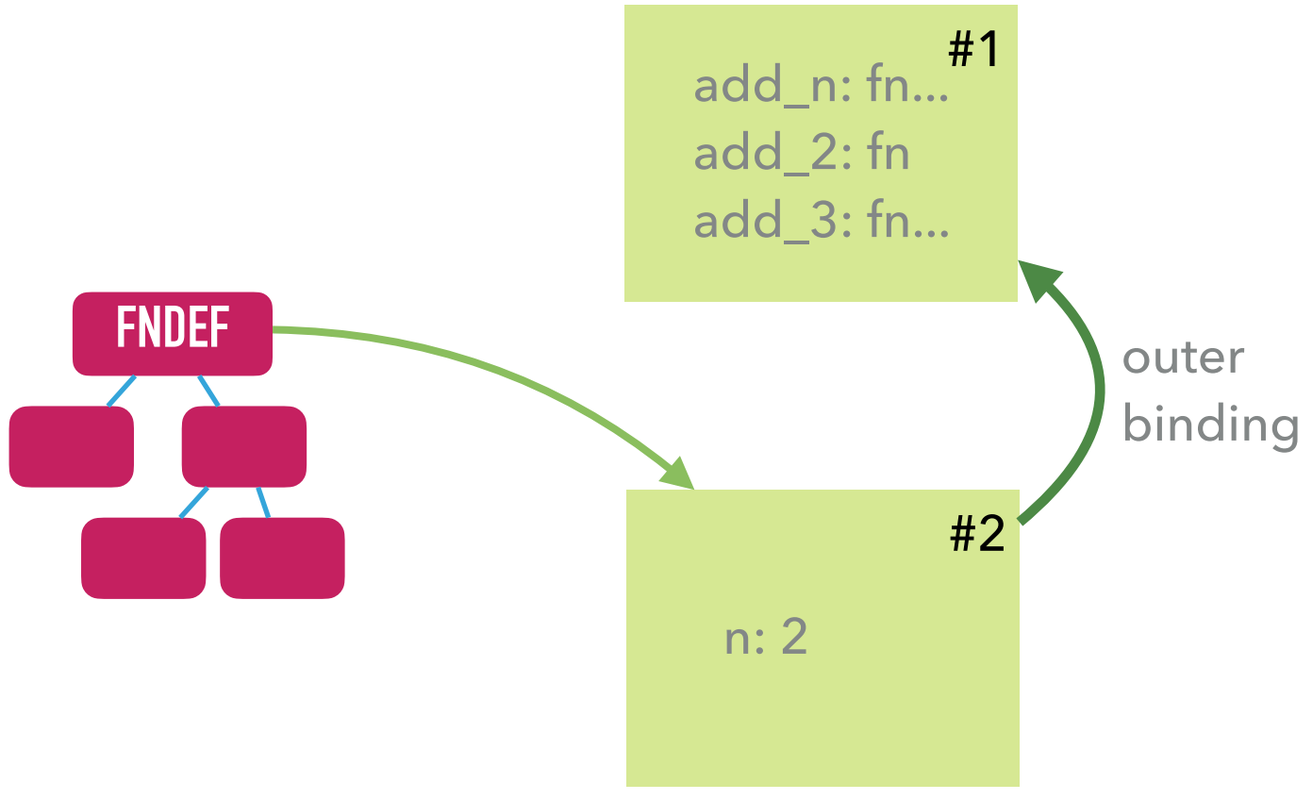


```
let add_n = fn (n) {
  fn (x) {
    x + n
  }
}

let add_2 = add_n(2)

let add_3 = add_n(3)

print(add_2(5))

print(add_3(5))
```



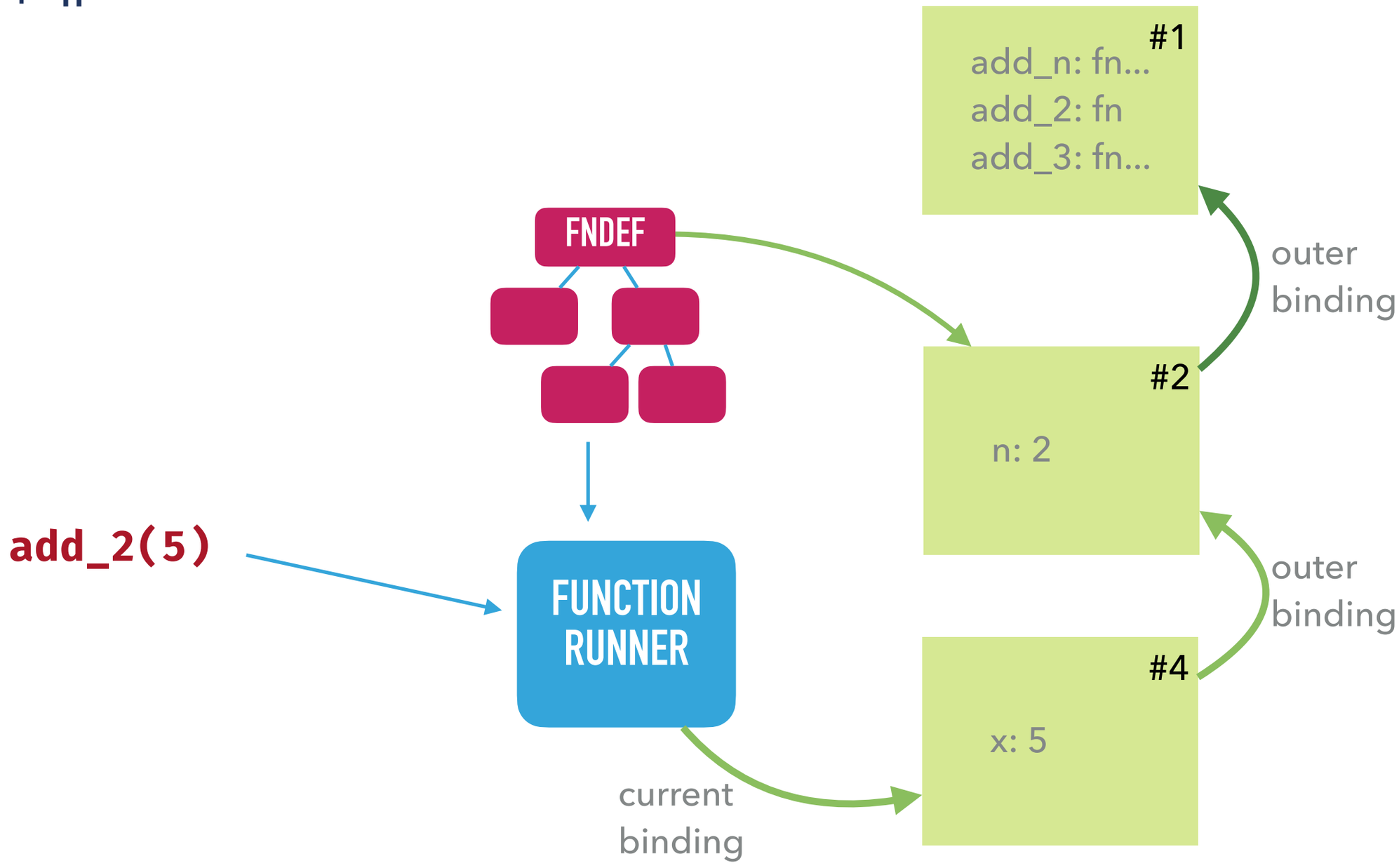
```
let add_n = fn (n) {
  fn (x) {
    x + n
  }
}

let add_2 = add_n(2)

let add_3 = add_n(3)

print(add_2(5))

print(add_3(5))
```



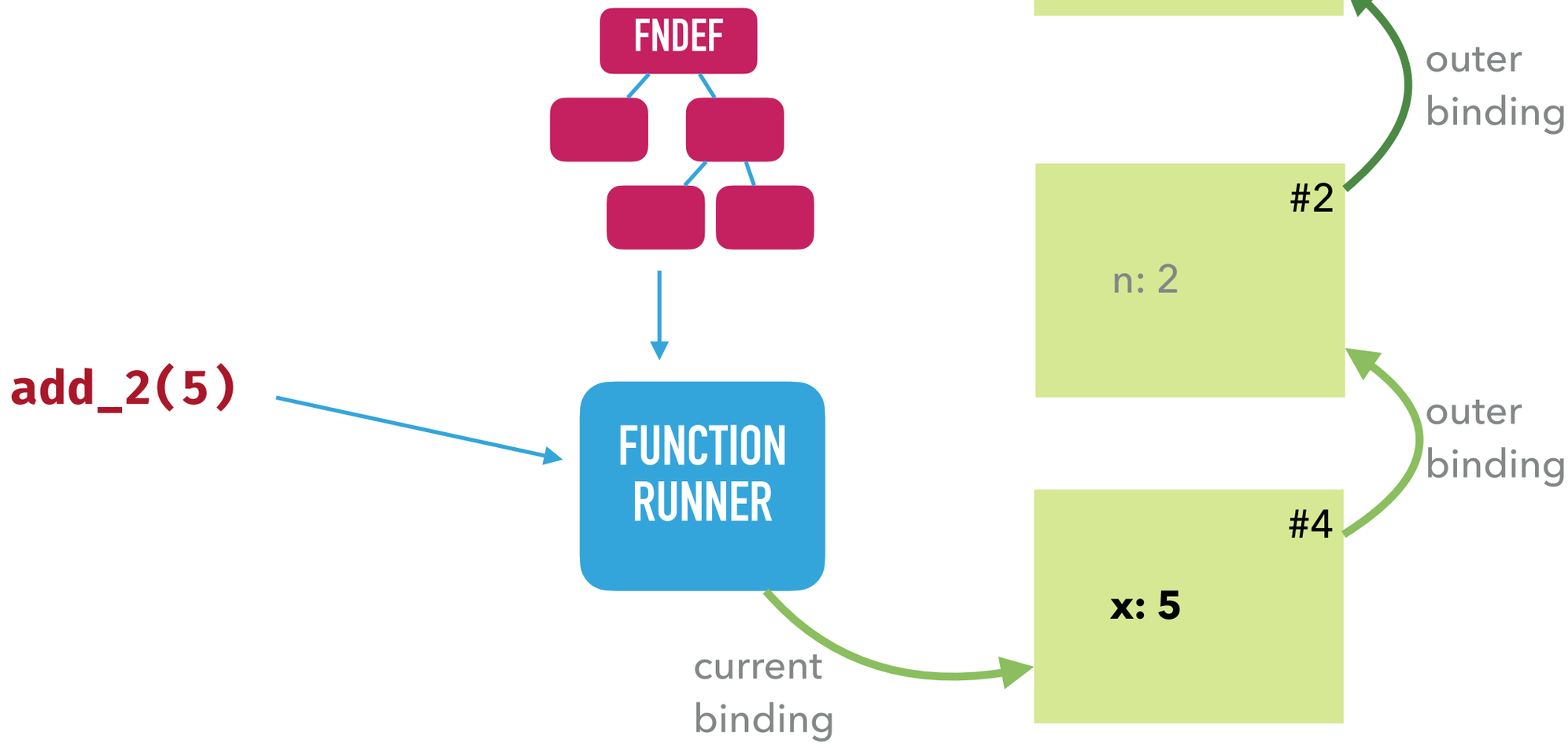



```
add_2(5)
```

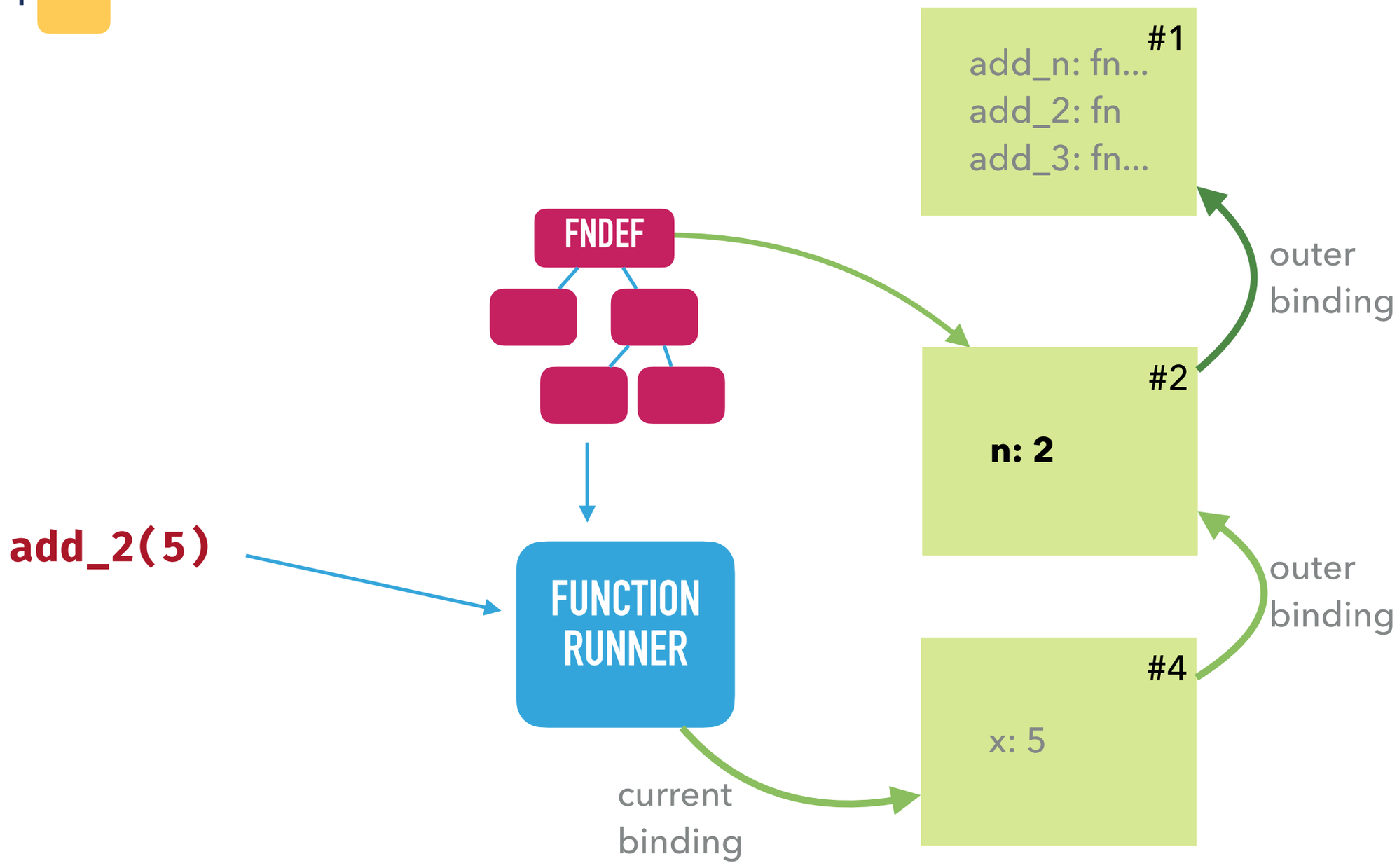
```
fn (x) {
  x + n
}
```

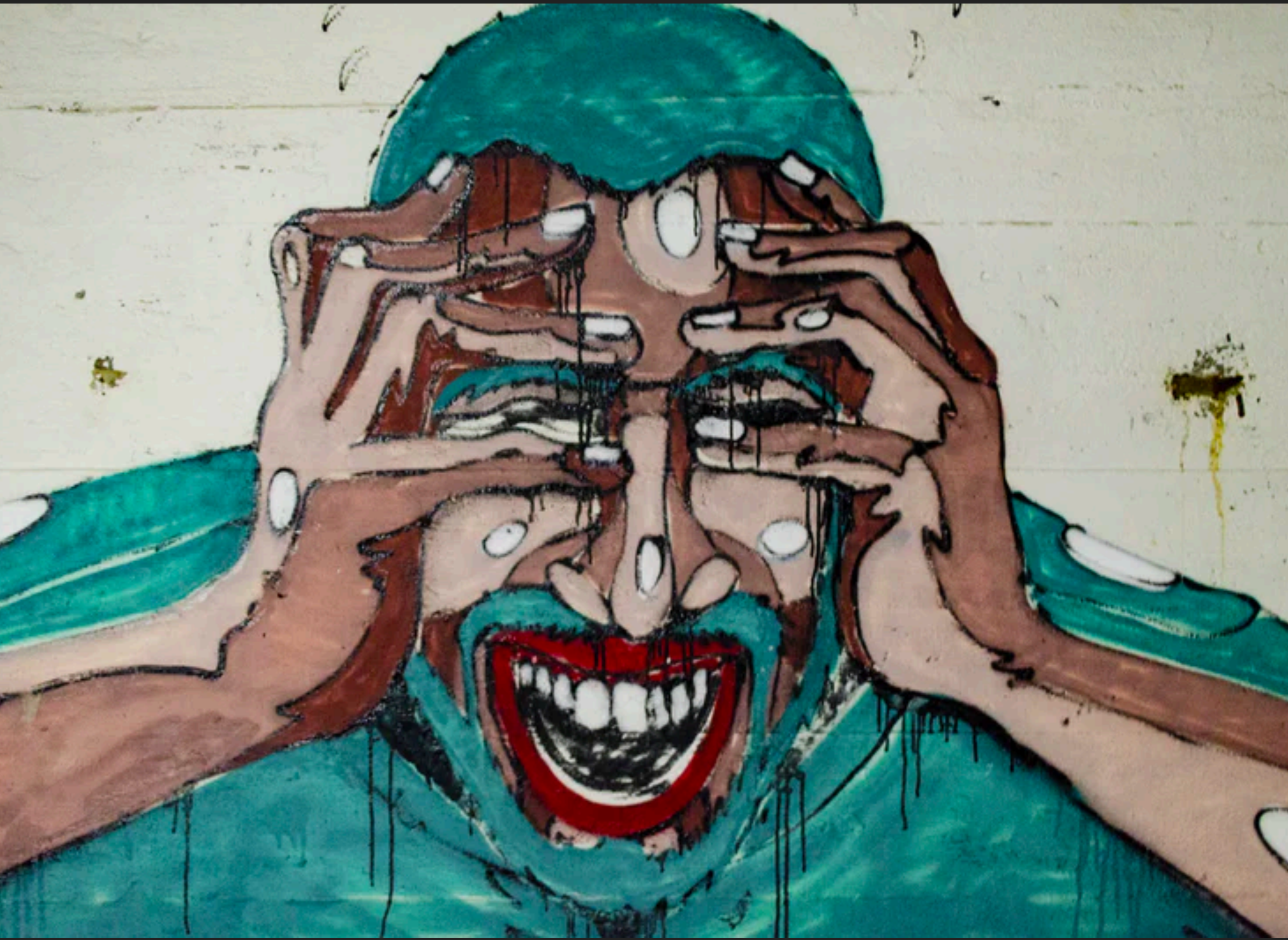

#1
add_n: fn...
add_2: fn
add_3: fn...
FNDEF
outer
binding
#2
n: 2
add_2(5)
FUNCTION
RUNNER
outer
binding
#4
x: 5
current
binding

```
fn (x) {
  x + n
}
```


FNDEF
add_2(5)
FUNCTION RUNNER
current binding
#1
add_n: fn...
add_2: fn
add_3: fn...
outer binding
#2
n: 2
outer binding
#4
x: 5

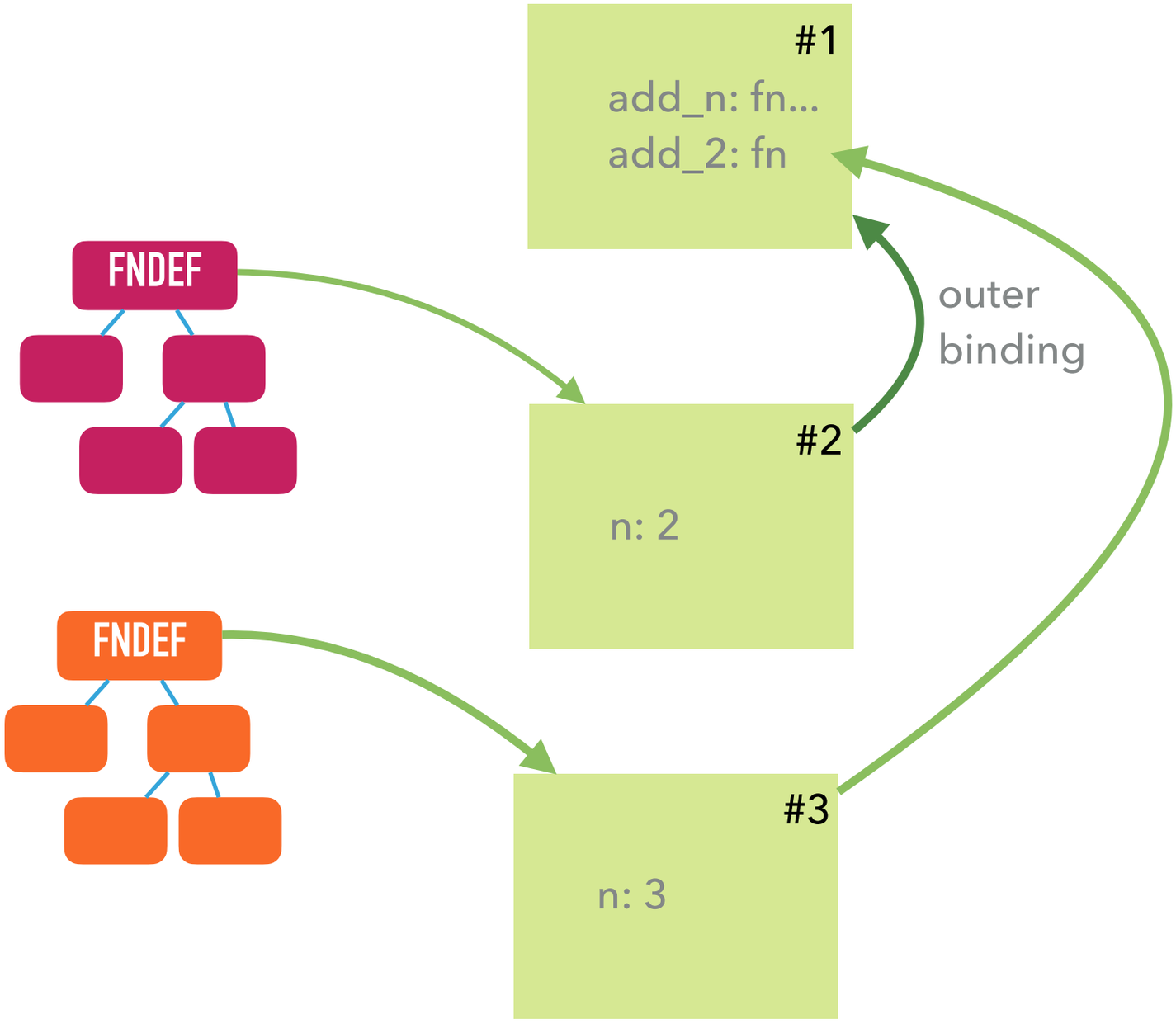
```
fn (x) {
   x +
}
```


#1
add_n: fn...
add_2: fn
add_3: fn...
FNDEF
outer
binding
#2
n: 2
add_2(5)
FUNCTION
RUNNER
outer
binding
#4
x: 5
current
binding



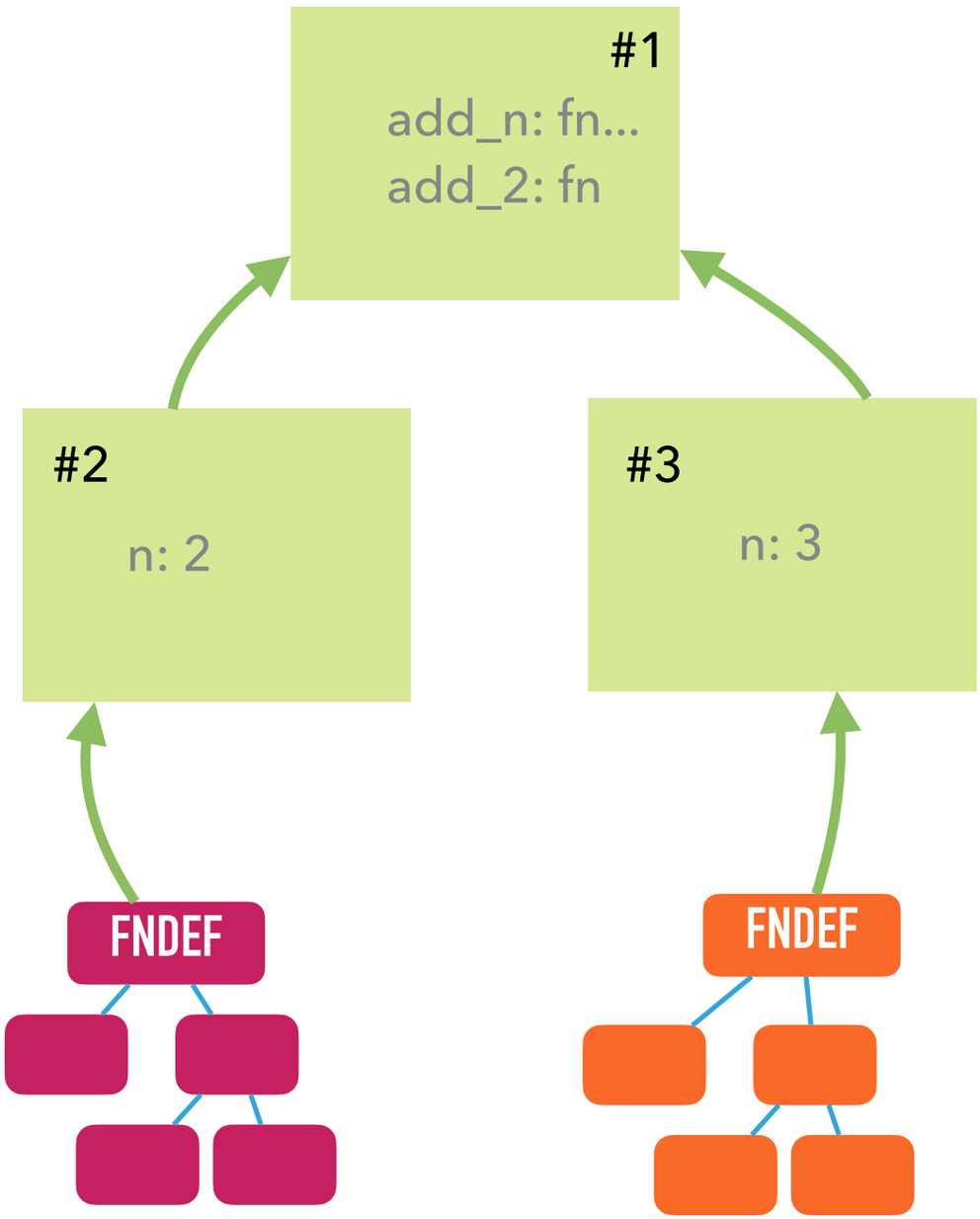
```
let add_n = fn (n) {
  fn (x) {
    x + n
  }
}

let add_2 = add_n(2)
let add_3 = add_n(3)
```

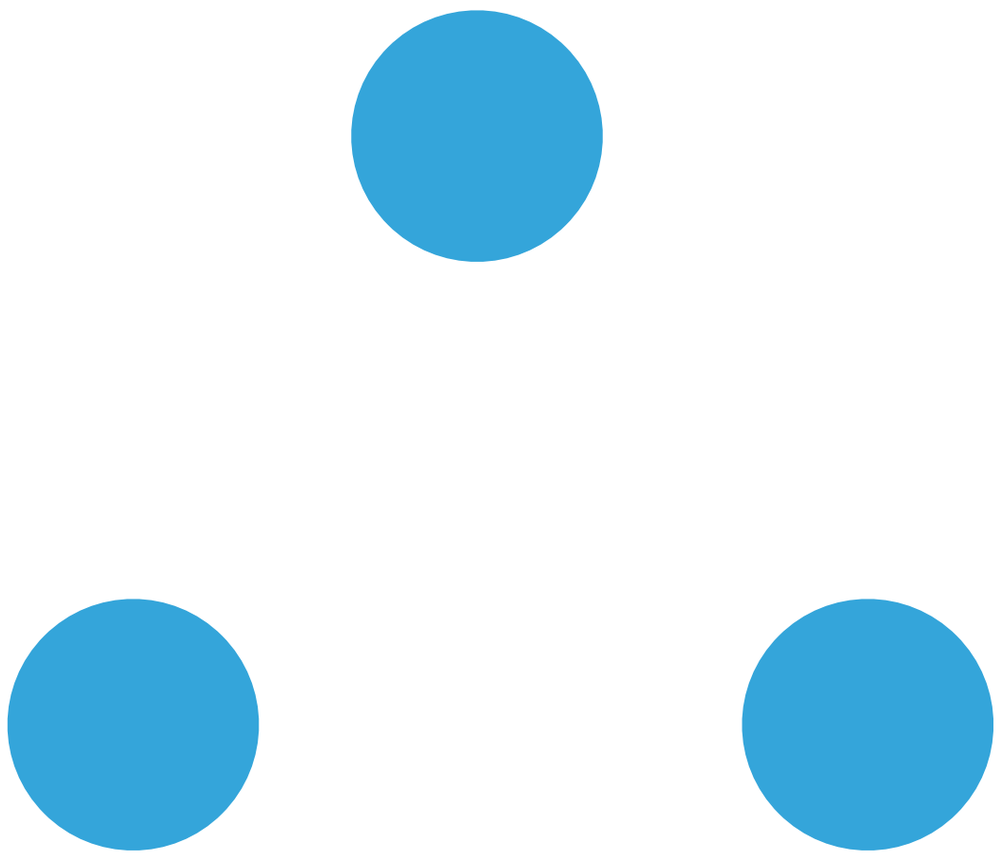
- one function definition in source code
- two different versions at runtime



```
let add_n = fn (n) {
  fn (x) {
    x + n
  }
}

let add_2 = add_n(2)
let add_3 = add_n(3)
```

- one function definition in source code
- two different versions at runtime





(therefore)

Executing a function definition creates a new runtime *thing* that can execute the function body in the appropriate binding.

runtime *thing* that can execute the function body in the appropriate binding.

THUNK

ast.js

```
export const FunctionDefinition =
               makeNode("FunctionDefinition", "formals", "code")


export const Thunk =
               makeNode("Thunk", "formals", "code", "binding")
```

This is the node added to the AST during the parse

This is the value returned when we execute a FnDecl

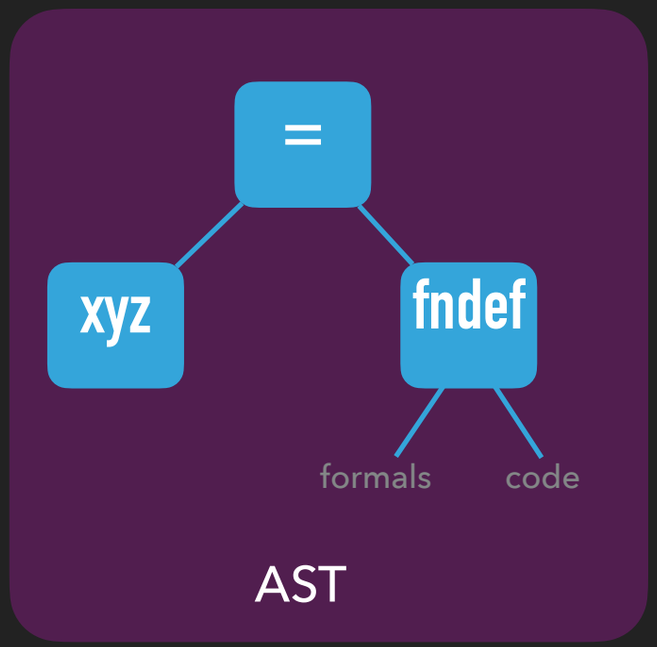
ast.js

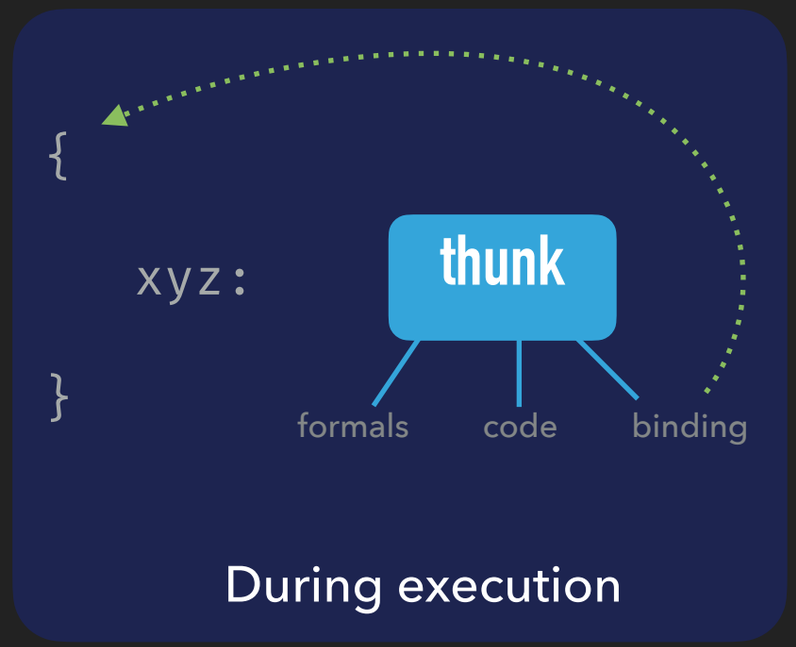
```
export const FunctionDefinition =
               makeNode("FunctionDefinition", "formals", "code")


export const Thunk =
               makeNode("Thunk", "formals", "code", "binding")
```

interpreter.js

```
 FunctionDefinition(node) {
    return new AST.Thunk(node.formals, node.code, this.binding)
 }
```

```
let xyz = fn () {  ...  }
```

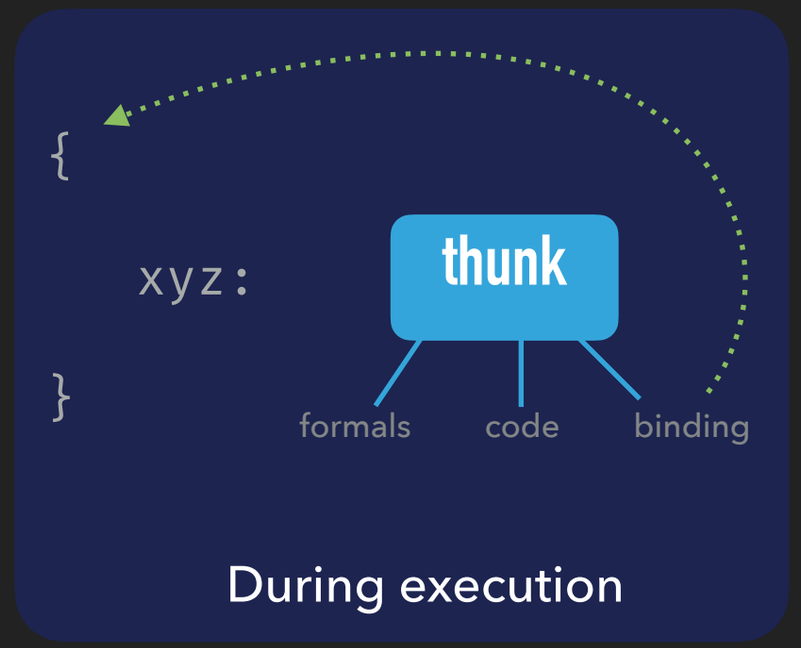




interpreter.js

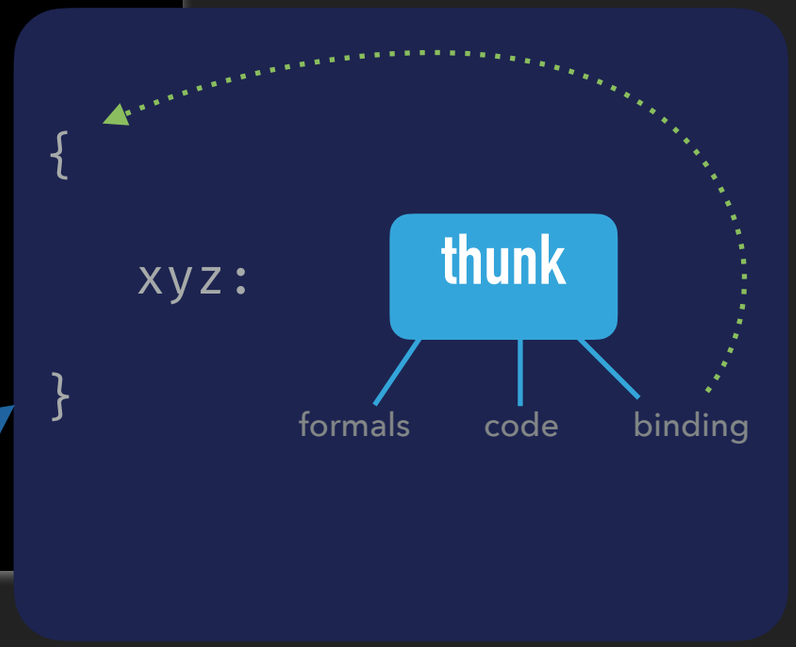
```
FunctionDefinition(node) {
   return new AST.Thunk(node.formals, node.code, this.binding)
}

FunctionCall(node) {
  let thunk = node.name.accept(this)
  // . . . argument handling



}
```

```
let xyz = fn (a) { a + 1 }

xyz(99)
```



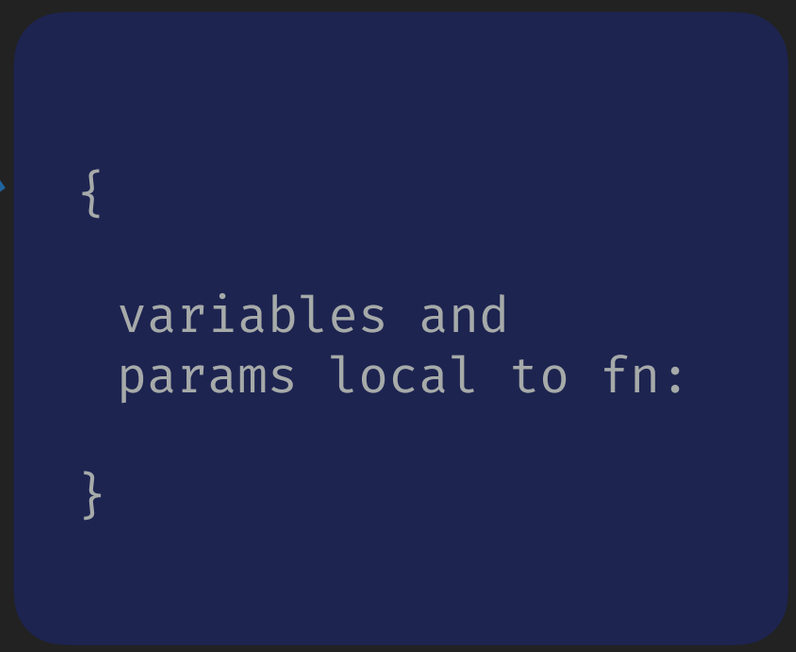
interpreter.js

```
FunctionDefinition(node) {
    return new AST.Thunk(node.formals, node.code, this.binding)
}

FunctionCall(node) {
  let thunk = node.name.accept(this)
  // . . . argument handling
  create new binding, whose parent is thunk.binding
  invoke thunk.code with that binding

}
```



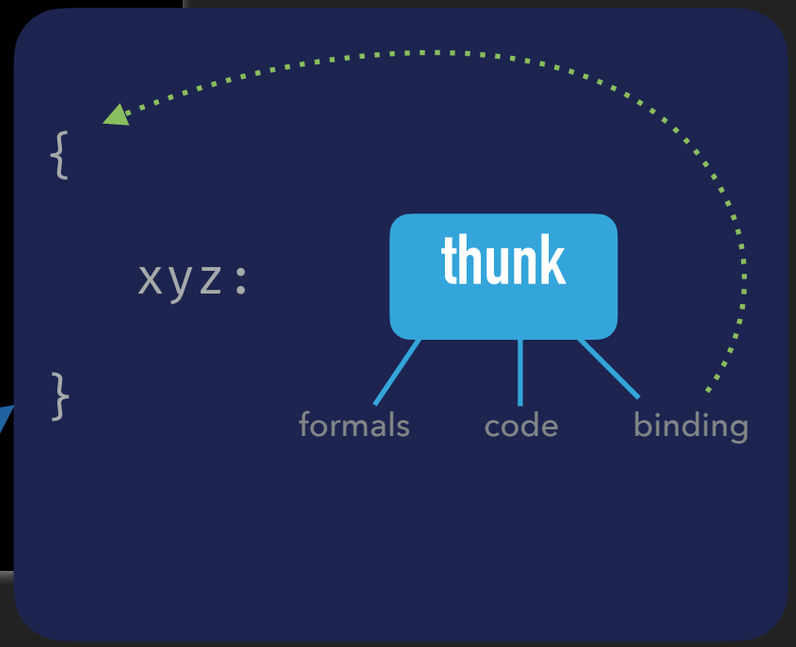
Parent binding

```
let xyz = fn (a) { a + 1 }

xyz(99)
```



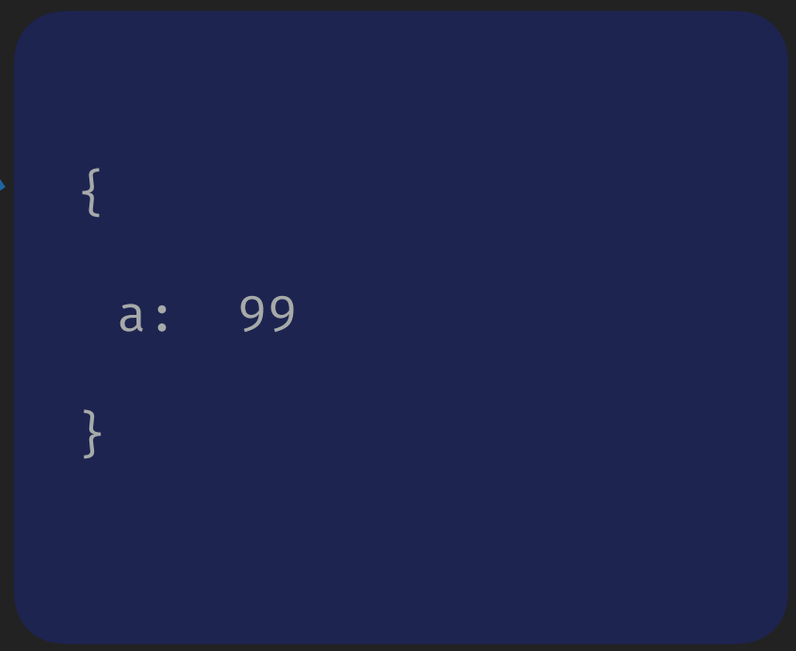
interpreter.js

```
FunctionDefinition(node) {
    return new AST.Thunk(node.formals, node.code, this.binding)
}

FunctionCall(node) {
  let thunk = node.name.accept(this)
  // . . . argument handling
  create new binding, whose parent is thunk.binding
  invoke thunk.code with that binding

}
```



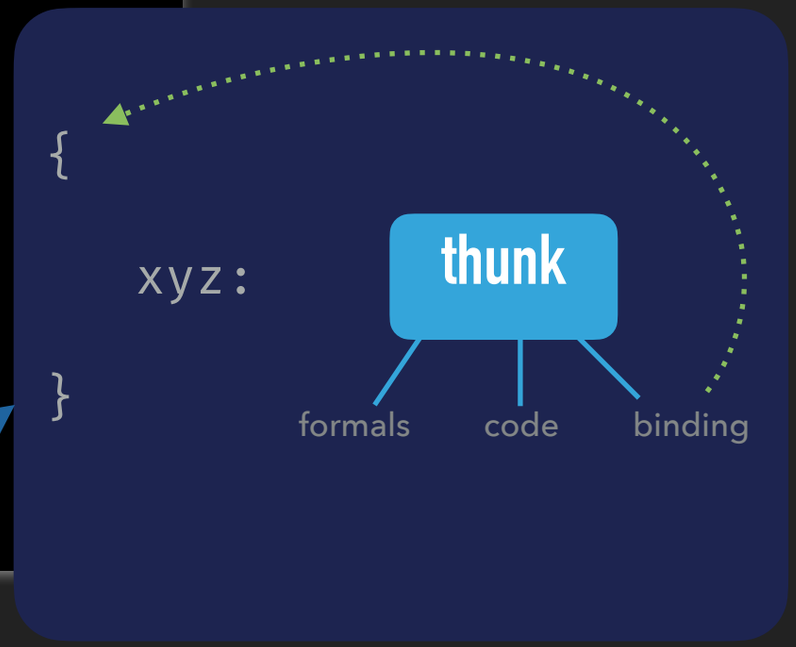
Parent binding

```
let xyz = fn (a) { a + 1 }

xyz(99)
```



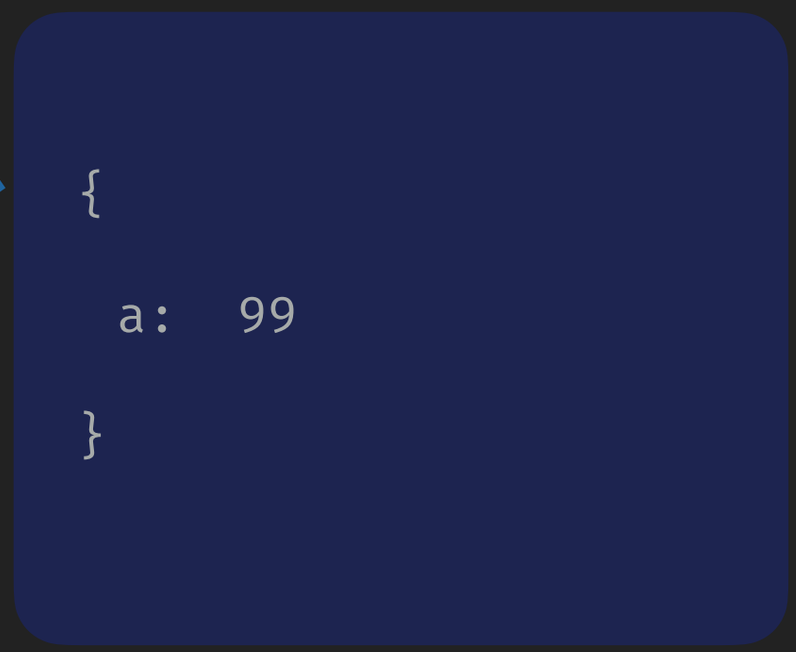
interpreter.js

```
FunctionDefinition(node) {
    return new AST.Thunk(node.formals, node.code, this.binding)
}

FunctionCall(node) {
  let thunk = node.name.accept(this)
  // . . . argument handling
  create new binding, whose parent is thunk.binding
  invoke thunk.code with that binding
  restore binding
}
```



Parent binding

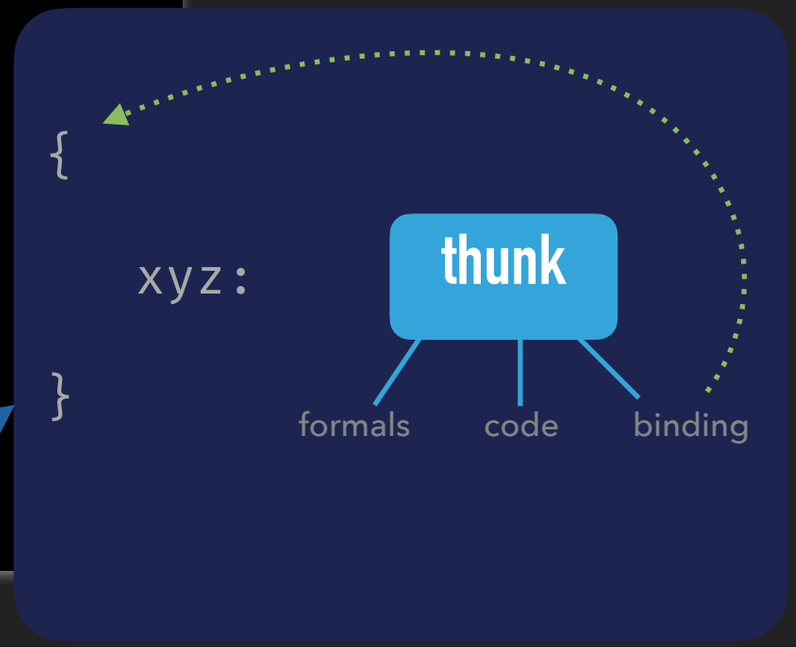
```
let xyz = fn (a) { a + 1 }

xyz(99)
```



What About the Parameters?

interpreter.js

```
FunctionDefinition(node) {
    return new AST.Thunk(node.formals, node.code, this.binding)
}

FunctionCall(node) {
  let thunk = node.name.accept(this)
  // . . . argument handling
  create new binding, whose parent is thunk.binding
  invoke thunk.code with that binding
  restore binding
}
```

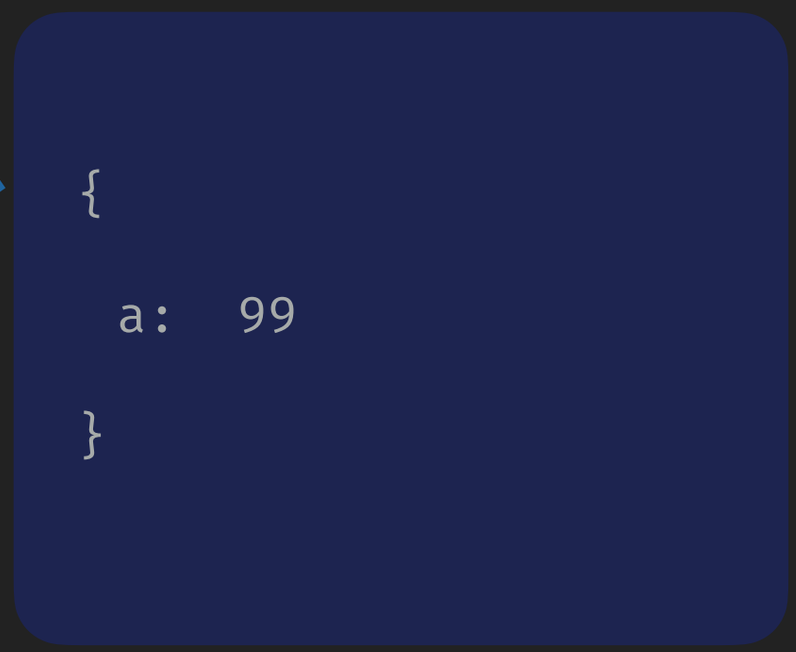



```
let xyz = fn (a) { a + 1 }
```

The thunk knows the parameter names

```
xyz(99)
```

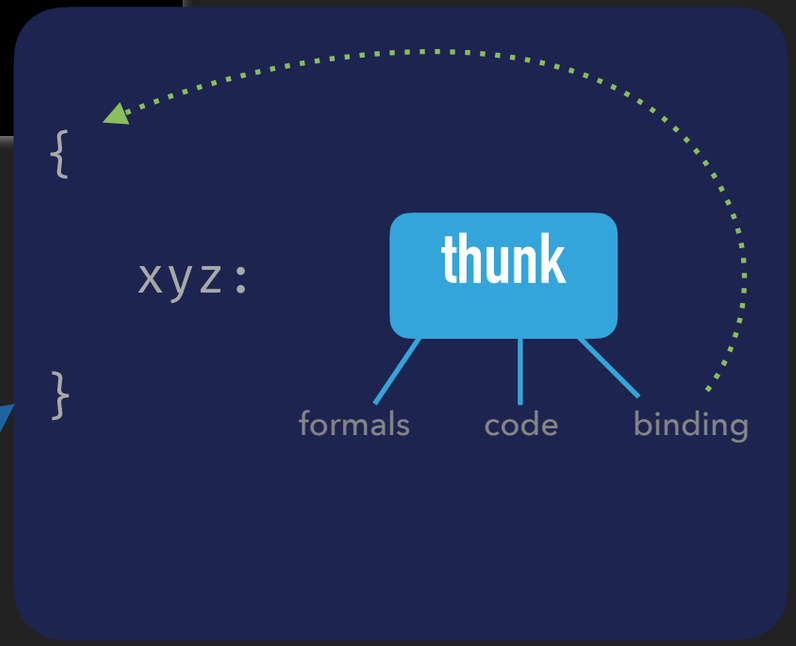
The call knows the values

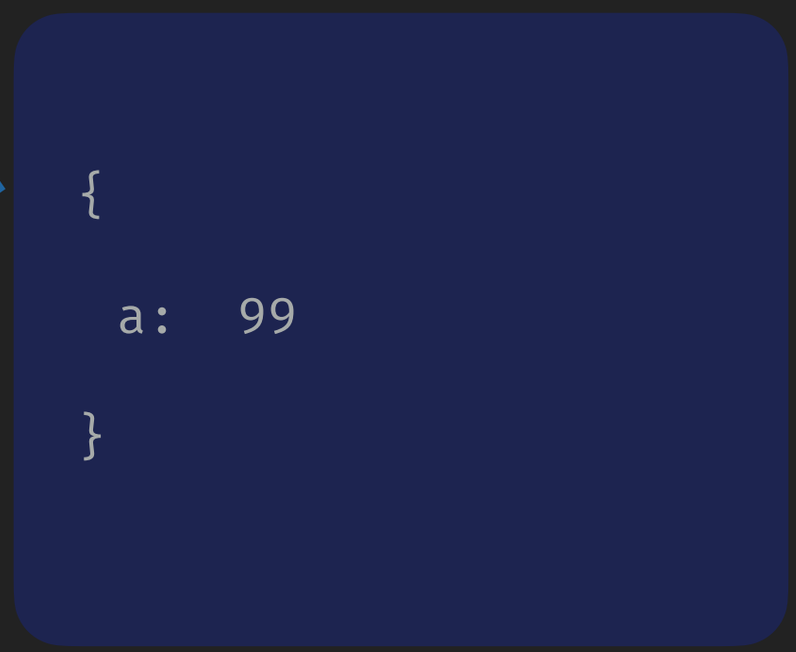


interpreter.js

```
FunctionCall(node) {
  let thunk = node.name.accept(this)
  // . . . argument handling
}
```

- convert node.args into a list of values
- verify same length as formals
- for each formal, set a variable with its name into the new binding, with the value from the corresponding arg
- (and variables declared in function body will automatically be stored in the new binding)





when we call a Thunk we

- Evaluate the actual parameters passed in the call
- Recover the binding in which the function was defined (stored in the thunk)
- Create a new context below that outer binding
- Match the name of each formal parameter to the corresponding value
- Store the name/value pair in the new, inner binding
- Execute the body of the function with that binding

What About the Bindings?

BINDINGS

- Previous our bindings was a global map.
- Now it's a tree
- Simplest implementation
  - each binding is a node containing a map
  - each binding contains a reference to its parent
    - (this is the opposite way around to trees you've built before)

class Binding

constructor(parent = null)

- remember parent
- create new map for binding

push()

- return new binding whose parent in this binding

pop()

- return parent of this binding

declareVariable(name, value)

- add or update this.binding with name/value

updateVariable(name, value)

- add or update this or parent bindings with name/value

getVariable(name)

- if this.binding has name, return value
- otherwise return parent.getBinding(name) (or error if no parent)

class Binding

declareVariable(name, value)

Always set in current binding

- add or update this.binding with name/value

updateVariable(name, value)

Lookup in all bindings

- add or update this or parent bindings with name/value

getVariable(name)

- if this.binding has name, return value
- otherwise return parent.getBinding(name) (or error if no parent)

```
let count = fn(by) {
  let c = 0
  fn() {
    c = c + by
  }
}
let byOnes = count(1)
let byTens = count(10)
print(byOnes(), byTens(), byOnes,  ... )
```